CS474 Project Midterm

Date: 2025-03-27 13:11

#HOMEWORK #CS474

Linear Models & SVM applied to Health Dataset

Content

Introduction

The purpose of this project is to both increase our knowledge of the implementations of linear models, support vector machine, and gradient descent, while applying them to a real world dataset.

Our choice of dataset is a heart disease csv found on Kaggle, which is a cumulation of three different health organization's data. Features included age, gender, resting heart rate, resting blood pressure, as well as numerous other nomial features, and a binary target feature indicating if that patient had heart disease.

Our focus is to implement each model using different subsets of features; linear regression model to predict a scalar value such as blood pressure using age and heart rate, perceptron model to determine if a patient has heart disease, logisitic regression to determine the probability of a patient having heart disease, and a support vector machine to compare the accuracy against the perceptron. Because we can't assume that our linear regression RSS will be invertible, gradient descent will also be implemented.

Our predicted outcomes are that the linear regression model may under perform due to the many other factors influencing biometrics, as well as not being able to fully utilize the nomial features, which will also effect the logisitic regression & perceptron for the same reasons (the underlying driver for each linear model is regression.) Due to this the support vector machine

may actually be a semi-useful predictor for heart disease, especially when compared to the perceptron.

Related Works

So far we have just been using the class notes and textbook; we will search for research papers for the final write up.

Methods

Data Processing

We have implemented our data processing functions utilizing `pandas` for efficient loading and indexing on these large `csv` files. So far we have implemented `load_features()` which extracts all samples & only the targeted features from the dataset, and `split_data()` which splits those samples into a training and testing set. In the near future we will improve `split_data()` to also create a validation set, or if we want to add better validation (such as cross validation), it may become a standalone function.

```
def load_features(filepath, features=["age", "sex", "resting bp s"], target=
["target"], remove=False):
```

This function takes the filepath and a set of features we will use to estimate the target feature. There is also an argument `remove` to trim any observation that contains null values for the features; this dataset, due to being three compiled sets, has many `0` values for some of the features that were not present in other datasets. The feature to look for `0` values is the first element in the feature list.

```
    data_df = pd.read_csv(filepath)
    if remove:  # remove 0 value observations for features[0]
        columns = features+target
        sub_data = data_df[columns]
        mask = sub_data[features[0]] != 0
        processed_data = sub_data[mask]
        features = np.asarray(processed_data[features])
        labels = np.asarray(processed_data[target])
    else:
        sub_data_df = data_df[features]
```

```
        labels_df = data_df[target]
        features = np.asarray(sub_data_df)
        labels = np.asarray(labels_df)
    return features, labels
```

The body of this function is fairly simple `pandas` indexing and operations, but it was new to us and a learning experience that was much more efficient that doing any other type of file io. `pandas` indexing was also more pythonic and efficient as we can use the column names as the indices, and even index all the features in one line by passing a list of the features we want: `sub_data_df = data_df[features]`.

```
def split_data(features, labels, reduce=False):
```

`split_data` simply slices the data into 2 portions:

```
    cutoff = math.floor(float(len(features)) * 0.75)
    trainX = features[:cutoff]
    testX = features[cutoff:]
    trainY = labels[:cutoff]
    testY = labels[cutoff:]
```

This current slicing may induce some bias, as well as being extremely large (and therefore computationally expensive) when performing simple testing (just making sure code compiles correctly.) Therefore we have added a `reduce` argument that reduces that dataset down to 100 total samples, using a random sampling with replacement (in python this just looks like generating random indices from the `rand` package.)

```
    if reduce: # trim sample to only 100 points
        trainX = []
        trainY = []
        testX = []
        testY = []
        for i in range(75):
            r = random.randint(0,len(features)-1)
            trainX.append(features[r])
            trainY.append(labels[r])
        for i in range(25):
            r = random.randint(0,len(features)-1)
            testX.append(features[r])
            testY.append(labels[r])
        return np.asarray(trainX), np.asarray(testX), np.asarray(trainY),
np.asarray(testY)
```

Linear Regression

```
def fit(self, X, y)
def accuracy(self, X, y)
def predict(self, X)
```

Our linear regression class is implemented by turning the math equations from our lecture notes directly into numpy statements:

- Representation : $\hat{y}_i = w^T x_i$
- Evaluation : $RSS(w) = (y - Xw)^T(y - Xw)$
- Optimization : $\frac{\partial t RSS(w)}{\partial w} = 0 \rightarrow w^* = (X^T X)^{-1} X^T y$

Representation is the `predict` function:

```
'''
    iterate through x and multiply by self.weights
    store y for each x and return list of y's
    y_i = w^T x_i
'''
return X @ self.weights
```

Evaluation is the `accuracy` function:

```
'''
    evaluates the model using RSS
    RSS(w) = (y - Xw)^t (y - Xw)
'''
residual = (y - (X @ self.weights))
rss = residual.T @ residual
print(f"RSS:\t{(rss[0,0])}")
tss = np.sum((y - np.mean(y)) ** 2)
r2_manual = 1 - (rss / tss)
print(f"R^2:\t{r2_manual[0,0]}")
return rss[0,0], r2_manual[0,0]
```

Optimization is the `fit` function:

```
'''
    finds the best fit line for the given X using dRSS(w)/dw
    w = (X^T X)^-1 X^T y
'''
x_t = X.T
```

```
inv = np.linalg.inv(x_t @ X)
self.weights = inv @ x_t @ y
```

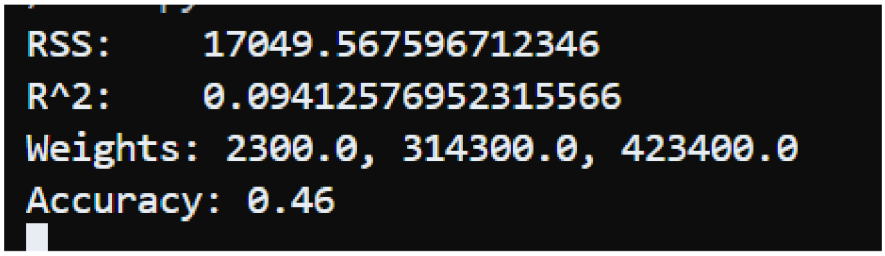
The current improvement to our model compared to previous homework assignments is graphing our regression in a higher dimensional space, as in previous homework assignments we only had 2 features.

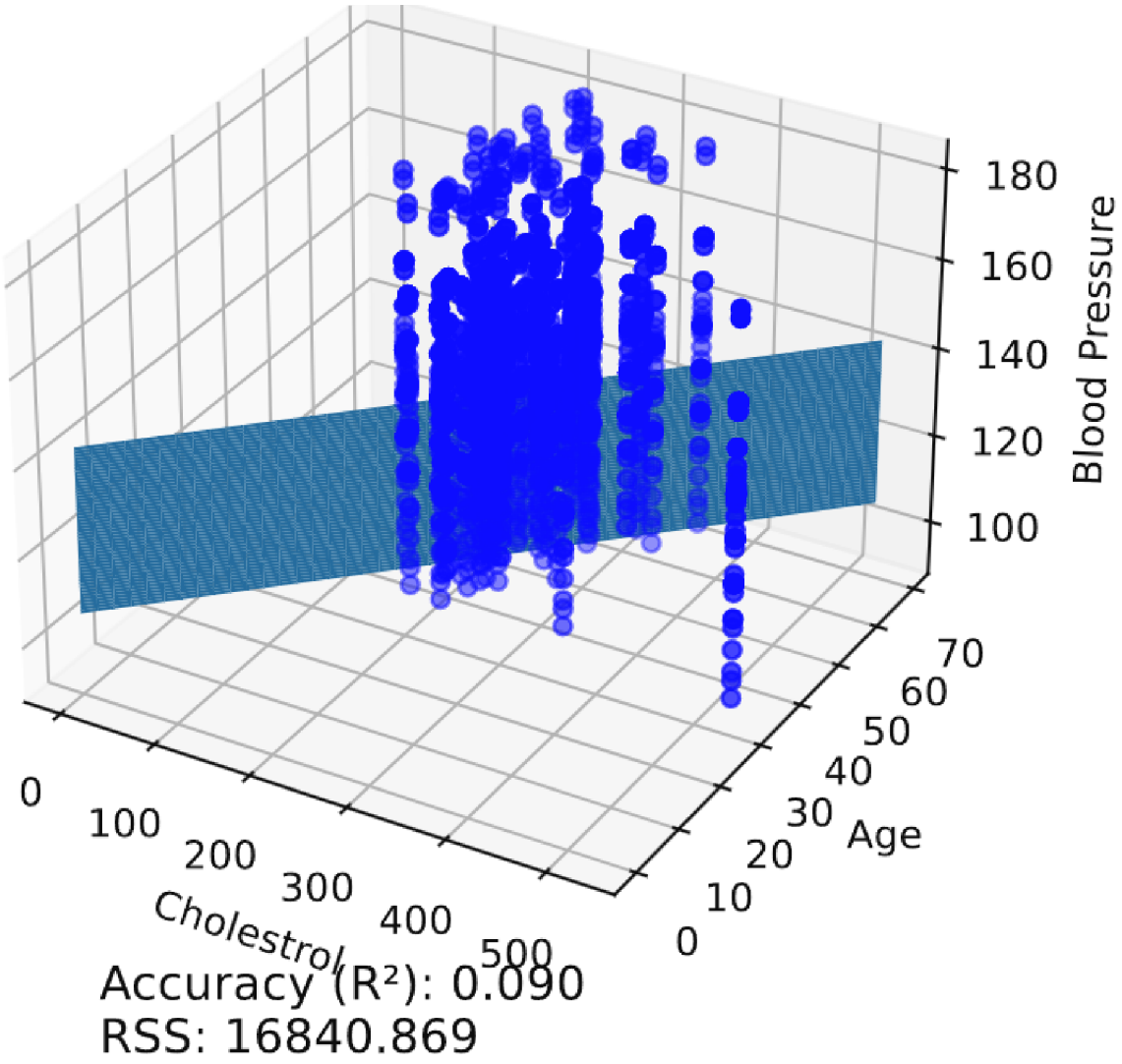
As noted in the introduction, we plan on implementing gradient descent in the near future.

Linear Perceptron

```
def fit(self, X, y)
def accuracy(self, X, y)
def predict(self, X)
```

Preliminary Results





We are still working on understanding and implementing 3D `maplotlib` graphs, so our graphs are not entirely useful (yet.)

Future Plan

Based on the very low accuracy from our regression model, we have two observations:

① The feature matrix may not be invertible
② The features may not be good predictors

To rule out if our features are not good predictors, we are currently finding a synthetic dataset to see if our regression model works as expected; if its able to get good results from a different dataset, then we know there are no bugs in our code. Once we know there are no bugs, then after implementing gradient descent we can compare the accuracy of this current model to the gradient descent model → if gradient descent provides considerably higher accuracy, then we know the issue was the invertibility, and the features are good enough predictors. If gradient descent still provides bad accuracy, then we can conclude that the features are just not enough to predict the target.

Once the regression model is validated & gradient descent is implemented, then implementing logistic regression will be very straightforward.

References

Yaser S. Abu-Mostafa, Malik Magdon-Ismail, Hsuan-Tien Lin, "Learning From Data" *AMLbook.com*
